EXHIBIT B

CASE Documents - Unverified English translation

<Basic Technology/Design>

Integrated CASE Development Environment for Embedded System

Kenji Numata* Akiko Okumura**

Abstract

[first paragraph of the Abstract not to be translated]

One of the features of embedded systems is that various actions according to each state are required when different events occur in the system. As a system has more and more combinations of states and events, it has an increasing number of bugs cause by failure in the specifications.

To solve this problem, there is a method of the State transition matrix. This paper describes the "Integrated CASE Environment" using "ZIPC," the CASE tool with the state transition matrix, into the current microcomputer software development environment.

1. Preface

In step with increases in the functionality of microcomputers that are used in embedded systems, the burden on developers has become large as software becomes enormous and complicated. In order to reduce this burden, software development environments are shifting from environments with only conventional assemblers to environments in which the C language can also be used. This is due to the fact that C compiler code efficiency has improved and the fact that environments in which debugging can be performed while looking at the C source have been established.

If an environment in which it were possible to directly generate efficient code from a design plan and perform debugging while looking at the design plan were provided, it would likely replace development using the C language.

2. "State Transition Matrix" Design Method

2.1 What is a state transition matrix?

A state transition matrix is a matrix in which the "processes" and "state transitions" that a system performs when certain "events" occur in certain "states" are described.

Figure 1 shows that if "event 1" occurs when in "state 1," the system executes "process 1" and moves into "state 2." Moreover, "/" in the matrix shows that processing is not performed, and "x" represents an unlikely combination. A transition destination number of "–" shows that a state transition is not made.

2.2 Features of state transition matrices

The following advantages are obtained by using state transition matrices.

(1) Creation of a simple design plan

A state transition diagram such as that which is shown in Figure 2 is a method of expressing the transition of "states." This does not yield problems in cases in which there are few combinations of states and events, but as the number of combinations increases, the arrows become intertwined and it becomes difficult to follow the condition of state transitions. However, with a matrix, the number of squares simply increases, so it is possible to simply read out state transitions (Figure 3).

[see source for figure: 1. "State transition matrix"]

* NEC Microcomputer Technology, Ltd., 2nd Applied Systems Department
** Microcomputer Division

[see source for figures:
2. "State transition diagram"
3. "Complexity state transition"
4. "Leveled state transition matrix"
5. "Expansive state transition matrix"]

(2) Prevention of specification failure/leaks

State transition matrices can include all of the combinations of "states" and "events," so it is possible to clearly define processes that are particularly easily overlooked. In Figure 1, in addition to the processes that are normally executed, "ignore" in the case in which processing is not performed and "impossible" in the case of an unlikely combination are also defined. By filling in all of the squares with some kind of content, it is possible to prevent specification failure/leaks.

2.3 Expanded and layered state transition matrices

While state transition matrices can indeed prevent failure/leaks, the matrix becomes enormous as the number of combinations of "states" and "events" increases. The following schemes are possible in cases in which matrices become too large.

(1) Layered state transition matrices

As shown in Figure 4, the lower matrices are defined with respect to each square of the matrix, and detailed information is given within these matrices. Matrix miniaturization and modularization are possible as a result of this scheme.

(2) Expanded state transition matrices

As shown in Figure 5, the cases of processes are divided in the state transition matrix by introducing variables into the matrix. As a result of this scheme, it is possible to reduce the number of "states" and simplify the matrix.

3. Development Work Flow and Tools Used

[see source for figures]

The development workflow for the Integrated CASE Environment introduced in this paper is shown in Figure 6 (a). The work flow is generally divided into 4 stages – "design plan (state transition matrix, panel image) creation," "design plan verification," "C source generation," and "source debugging."

As shown in Figure 6 (b), a hardware configuration in which the host machine and IE (incircuit emulator) are connected to the user system is used.

4. Work Sequence

4.1 Design plan creation

(1) Panel image creation

The panel image shown in Figure 7 (a) is created. Using the input/output panel of the simulator SM78K0, elements such as buttons, LED's, and speakers are arranged in a window, and ports corresponding to the target chip are assigned to each part. This panel can be used for the confirmation of the hardware completion schedule as well as the verification of the operations of later processes.

[see source for figures:
6. "Development process and using tools"
7. "Inspection of specification"]

(2) State transition matrix creation

The operations and processes of the system are considered from the panel image that is created. The types of "states" and "events" in the system are defined. The defined contents are written together with the "processes" with a ZIPC* editor, and a state transition matrix such as that which is shown in Figure 7 (b) is thus created.

4.2 Design plan verification

When the panel image and state transition matrix are complete, the prototype operations are verified using the input/output panel of the simulator SM78K0 and a ZIPC simulator. Using the windows shown in Figure 7 (a) and (b), the process execution contents corresponding to events inputted from the panel image are confirmed in the state transition matrix.

By performing sufficient verification operations at this stage, it is possible to sort out bugs in the design plan.

4.3 C source generation

The C source is automatically generated with the ZIPC simulator from the state transition matrix verified with the ZIPC simulator. Basically, the combinations of "states" and "events" are analyzed, and a skeleton of the source up until "processes" are called is generated. However, if the "process" portion is also created in accordance with the ZIPC rules, then a C source that does not require retouching is automatically generated.

4.4 Source debugging while looking at the matrix

With conventional environments, an IE (incircuit emulator) was connected to the host machine and debugging was performed while looking at the source on a debugger. With the Integrated CASE Environment, as shown in Figure 8, by using a ZIPC emulator and the debugger ID78K0, debugging is performed while looking at the state transition matrix in addition to the conventional C source debugging environment. It is possible to perform efficient debugging by switching between the two methods – the state transition matrix method and the C source method – depending on the situation.

* ZIPC is a product of the Tesco Company (Inc.)

[see source for figure: 8. "Debug Image" and table: "Result of apply."]

5. Example of Application

We actually developed a demo set (telephone set) with this development environment. Code size comparisons with the case in which the demo set was developed using a flow chart and an assembler with the conventional environment are shown in the table. Although there was the diversion of the driver part, results indicating that the code size was almost exactly the same as the code size of the conventional design were obtained.

6. Conclusion

In this paper we introduced the Integrated CASE Environment for microcomputer software. In order to restrain increases in development man-hours due to increases in the scale of software, it is necessary to reduce bugs in later processes by sufficiently performing verification operations at the design stage. The environment introduced in this paper enables the dynamic verification of design plans and allows debugging while looking at the design plans, thus making it possible to reduce the burden on software developers. This development environment could potentially replace current development environments.

This environment is presently provided for the 8-bit single chip microcomputer 78K/0 series, and there are plans to expand to the 16-bit 78K/4 and 32-bit RISC V850. In addition to expansion to other devices, there are also plans for further expansion of state transition matrices and enhancement of simulator functionality.

[remainder not to be translated]